UNITED STATES DESIGN PATENT APPLICATION

FOR

METHOD AND APPARATUS FOR FACILITATING RECOGNITION OF AN OPEN EVENT WINDOW DURING OPERATION OF GUEST SOFTWARE IN A VIRTUAL MACHINE ENVIRONMENT

Inventors:

STEVEN M. BENNETT

ANDREW V. ANDERSON

ERIK COTA-ROBLES

STALINSELVARAJ JEYASINGH

ALAIN KÄGI

GILBERT NEIGER

RICHARD UHLIG

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard, Seventh Floor
Los Angeles, CA 90025-1026

(408) 720-8300

EXPRESS MAIL CERTIFICATE OF MAILING

"Express Mail" mailing label number __EV409361118US__

Date of Deposit __March 31, 2004__

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Mail Stop Patent Application, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450

_____Michelle Begay_____

(Typed or printed name of person mailing paper or fee)

_____

(Signature of person mailing paper or fee)

METHOD AND APPARATUS FOR FACILITATING RECOGNITION OF AN OPEN EVENT WINDOW DURING OPERATION OF GUEST SOFTWARE IN A VIRTUAL MACHINE ENVIRONMENT

Field

[0001] Embodiments of the invention relate generally to virtual machines, and more specifically to facilitating recognition of an open event window during operation of guest software in a virtual machine environment.

Background of the Invention

[0002] A closed interrupt window is a time interval during software execution when interrupts may not be delivered to the software. Alternatively, an open interrupt window is a time interval during software execution when interrupts can be delivered to the software. The opening and closing of the interrupt window may occur through a number of mechanisms.

[0003] The interrupt window may be closed because the software does not want to be interrupted. For example, an instruction set architecture (ISA) may allow software to block external interrupts through a masking bit or some other mechanism. In particular, in the ISA of the Intel® Pentium® 4 (referred to herein as the IA-32 ISA), hardware interrupts are blocked if the IF bit in the EFLAGS register is cleared. The IF bit may be set or cleared by instructions that load the EFLAGS register (e.g., POPF) or by instructions that explicitly set or clear the IF bit (e.g., STI, CLI). Additionally, certain machine transitions may make modifications to the IF bit (e.g., interrupt and exception vectoring may cause the IF bit to be cleared).

[0004] The interrupt window may be closed because the software is in the midst of a transition in machine state that precludes delivering the interrupt. In the IA-32 ISA, for example, because the proper vectoring of an interrupt requires the stack segment and stack pointer to remain consistent, the CPU may automatically block interrupt vectoring while the software update of the stack segment and stack pointer (MOVSS/POPSS) takes place. Additionally, the IA-32 ISA dictates that the execution of an STI instruction that sets the IF bit does not take effect until the instruction following the STI completes. Hence, the interrupt window opens one instruction after the STI.

[0005] Additional events, for example, non-maskable interrupts (NMIs), system management interrupts (SMIs), etc. may have similar event blocking semantics. A closed event window is a time interval during software execution when the associated event may not be delivered to the software. Alternatively, an open event window is a time interval during software execution when the associated event can be delivered to the software.

[0006] Depending on the specific ISA, the event window conditions may be rather complex, requiring evaluation of detailed information on the state of the processor to determine if software has an open event window for a given event.

Brief Description of the Drawings

[0007] The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

[0008] **Figure 1** illustrates one embodiment of a virtual-machine environment, in which the present invention may operate;

[0009] **Figure 2** is a flow diagram of one embodiment of a process for facilitating recognition of an open event window during operation of guest software;

[0010] **Figure 3** illustrates simplified operation of reorder buffer logic, according to one embodiment of the present invention;

[0011] **Figure 4** is a flow diagram of one embodiment of a process for determining the status of an event window of a VM; and

[0012] **Figure 5** is a flow diagram of one embodiment of a process for delivering an event to guest software.

Description of Embodiments

[0013] A method and apparatus for facilitating recognition of an open event window during operation of guest software is described. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention can be practiced without these specific details.

[0014] Some portions of the detailed descriptions that follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer system's registers or memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of operations leading to a desired result. The operations are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[0015] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically

stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or the like, may refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer-system memories or registers or other such information storage, transmission or display devices.

[0016] In the following detailed description of the embodiments, reference is made to the accompanying drawings that show, by way of illustration, specific embodiments in which the invention may be practiced. In the drawings, like numerals describe substantially similar components throughout the several views. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention. Other embodiments may be utilized and structural, logical, and electrical changes may be made without departing from the scope of the present invention. Moreover, it is to be understood that the various embodiments of the invention, although different, are not necessarily mutually exclusive. For example, a particular feature, structure, or characteristic described in one embodiment may be included within other embodiments. The following detailed description is, therefore, not to be taken in a limiting sense, and the

scope of the present invention is defined only by the appended claims, along with the full scope of equivalents to which such claims are entitled.

[0017] Although the below examples may describe detection of an open event window during operation of a virtual machine in the context of execution units and logic circuits, other embodiments of the present invention can be accomplished by way of software. For example, in some embodiments, the present invention may be provided as a computer program product or software which may include a machine or computer-readable medium having stored thereon instructions which may be used to program a computer (or other electronic devices) to perform a process according to the present invention. In other embodiments, steps of the present invention might be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

[0018] Thus, a machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, a transmission over the Internet, electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.) or the like.

[0019] Further, a design may go through various stages, from creation to simulation to fabrication. Data representing a design may represent the design in a number of manners. First, as is useful in simulations, the hardware may be represented using a hardware description language or another functional description language. Additionally, a circuit level model with logic and/or transistor gates may be produced at some stages of the design process. Furthermore, most designs, at some stage, reach a level of data representing the physical placement of various devices in the hardware model. In the case where conventional semiconductor fabrication techniques are used, data representing a hardware model may be the data specifying the presence or absence of various features on different mask layers for masks used to produce the integrated circuit. In any representation of the design, the data may be stored in any form of a machine-readable medium. An optical or electrical wave modulated or otherwise generated to transmit such information, a memory, or a magnetic or optical storage such as a disc may be the machine readable medium. Any of these mediums may "carry" or "indicate" the design or software information. When an electrical carrier wave indicating or carrying the code or design is transmitted, to the extent that copying, buffering, or re-transmission of the electrical signal is performed, a new copy is made. Thus, a communication provider or a network provider may make copies of an article (a carrier wave) embodying techniques of the present invention.

[0020] **Figure 1** illustrates one embodiment of a virtual-machine environment 100, in which the present invention may operate. In this embodiment, bare platform hardware 116 comprises a computing platform, which may be capable, for example, of executing a standard operating system (OS) or a virtual-machine monitor (VMM), such as a VMM 112.

[0021] The VMM 112, though typically implemented in software, may emulate and export a bare machine interface to higher level software. Such higher level software may comprise a standard or real-time OS, may be a highly stripped down operating environment with limited operating system functionality, may not include traditional OS facilities, etc. Alternatively, for example, the VMM 112 may be run within, or on top of, another VMM. VMMs may be implemented, for example, in hardware, software, firmware or by a combination of various techniques.

[0022] The platform hardware 116 can be of a personal computer (PC), mainframe, handheld device, portable computer, set-top box, or any other computing system. The platform hardware 116 includes a processor 118 and memory 120.

[0023] Processor 118 can be any type of processor capable of executing software, such as a microprocessor, digital signal processor, microcontroller, or the like. The processor 118 may include microcode, programmable logic or hardcoded logic for performing the execution of method embodiments of the present invention. Although **Figure 1** shows only one such processor 118, there may be one or more processors in the system.

[0024] Memory 120 can be a hard disk, a floppy disk, random access memory (RAM), read only memory (ROM), flash memory, any combination of the above devices, or any other type of machine medium readable by processor 118. Memory 120 may store instructions and/or data for performing the execution of method embodiments of the present invention.

[0025] The VMM 112 presents to other software (i.e., "guest" software) the abstraction of one or more virtual machines (VMs), which may provide the same or different abstractions to the various guests. **Figure 1** shows two VMs, 102 and 114. The guest software running on each VM may include a guest OS such as a guest OS 104 or 106 and various guest software applications 108 and 110. Each of the guest OSs 104 and 106 expect to access physical resources (e.g., processor registers, memory and I/O devices) within the VMs 102 and 114 on which the guest OS 104 or 106 is running and to perform other functions. For example, the guest OS expects to have access to all registers, caches, structures, I/O devices, memory and the like, according to the architecture of the processor and platform presented in the VM. The resources that can be accessed by the guest software may either be classified as "privileged" or "non-privileged." For privileged resources, the VMM 112 facilitates functionality desired by guest software while retaining ultimate control over these privileged resources. Non-privileged resources do not need to be controlled by the VMM 112 and can be accessed by guest software.

[0026] Further, each guest OS expects to handle various events such as exceptions (e.g., page faults, general protection faults, etc.), interrupts (e.g.,

hardware interrupts, software interrupts), and platform events (e.g.,

initialization (INIT) and system management interrupts (SMIs)). These

exceptions, interrupts and platform events are referred to collectively and

individually as "events" herein. Some of these events are "privileged"

because they must be handled by the VMM 112 to ensure proper operation of

VMs 102 and 114 and for protection from and among guest software.

[0027] When a privileged event occurs or guest software attempts to

access a privileged resource, control may be transferred to the VMM 112. The

transfer of control from guest software to the VMM 112 is referred to herein as

a VM exit. After facilitating the resource access or handling the event

appropriately, the VMM 112 may return control to guest software. The

transfer of control from the VMM 112 to guest software is referred to as a VM

entry.

[0028] In one embodiment, the processor 118 controls the operation of

the VMs 102 and 114 in accordance with data stored in a virtual machine

control structure (VMCS) 124. The VMCS 124 is a structure that may contain

state of guest software, state of the VMM 112, execution control information

indicating how the VMM 112 wishes to control operation of guest software,

information controlling transitions between the VMM 112 and a VM, etc. The

processor 118 reads information from the VMCS 124 to determine the

execution environment of the VM and to constrain its behavior. In one

embodiment, the VMCS is stored in memory 120. In some embodiments,

multiple VMCS structures are used to support multiple VMs.

[0029] When the VMM 112 receives control following an event such as, for example, an interrupt (e.g., an interrupt occurred during the operation of the VMM 112 or an interrupt occurred during operation of a VM, resulting in a VM exit), the VMM 112 may handle the event itself or decide that the event should be handled by an appropriate VM. If the event should be handled by a VM, it may only be delivered when the VM is ready to receive the event (e.g., if the event is an interrupt, the VM has an open interrupt window). In one embodiment, the VMM 112 includes a pending event module 130 that is responsible for determining whether the VM's relevant event window is open and, if the relevant event window is not open, delaying the delivery of the event to the VM until the relevant event window opens.

[0030] In an embodiment, the pending event module 130 determines that the event window has opened upon receiving an indication from the processor 118. In one embodiment, the pending event module 130 uses the indication of the event window status to decide whether to deliver a pending event which requires an open event window, to the VM. In another embodiment, the pending event module 130 uses the indication of the event window status to facilitate functionality other than delivery of pending events which require an open event window. For example, the pending event module 130 may utilize information regarding the status of a VM's event window to make VM scheduling decisions (e.g., the VMM may choose not to switch to another VM at a scheduling point if the current VM has a closed interrupt window). Rather, the pending event module 130 may utilize the

pending event indicator to generate a VM entry to the VM that will execute until the VM has an open event window, at which time the VMM 112 may choose to schedule another VM to execute.

[0031] In one embodiment, the processor 118 includes event window monitoring logic 122 that provides such an indication to the pending event module 130. In particular, once the pending event module 130 determines that the event window of the VM is closed, it sets a pending event indicator to a delivery value and requests a VM entry to this VM. In one embodiment, the pending event indicator is stored in the VMCS 124. There may be one or more such pending event indicators associated with one or more events. For example, the VMCS 124 may include a pending interrupt indicator and a pending SMI indicator, corresponding to interrupt and SMI events, respectively. Alternatively, the pending event indicator(s) may reside in the processor 118, a combination of the memory 120 and the processor 118, or in any other storage location or locations.

[0032] In response to the VM entry request, the event window monitoring logic 122 transitions control to the VM and starts monitoring the VM's relevant event window(s). Upon detecting that the VM's relevant event window is open, the event window monitoring logic 122 transfers control back to the VMM 112. In one embodiment, the event window monitoring logic 122 notifies the VMM 112 (e.g., using a designated reason code to indicate the source of the VM exit) that the VM exit was caused by an open

event window. In an embodiment, there may be one such designated reason code for each event type.

[0033] In one embodiment, the event window monitoring logic 122 begins monitoring the state of the VM's event window upon executing a VM entry request issued by the VMM 112 and determining that the VM entry request is associated with a request to be informed of an open event window. In one embodiment, the event monitoring logic 122 determines whether the VM entry request is associated with a request to be informed of an open event window based on a current value of the pending event indicator. There may be one or more such pending event indicators, each corresponding to one or more events. For example, in an embodiment, there may be a pending interrupt indicator corresponding to interrupts and a pending SMI indicator corresponding to SMIs.

[0034] In one embodiment, the determination of whether the event window is open for interrupts is based on a current value of an interrupt flag that can be updated by guest software when the state of guest software's ability to accept interrupts changes. For example, in the IA-32 ISA, the EFLAGS register contains the IF interrupt flag bit, which, in part, controls whether an interrupt will be delivered to the software (as discussed above, in the IA-32 ISA, other factors that may block interrupts are considered in determining if the interrupt window is open). Alternatively, any other mechanism known in the art can be used to determine whether the interrupt

window of the VM is open. Other events may have similar mechanisms for determining the state of the associated event window.

[0035] Once the VMM 112 receives an indication that the event window of the VM is open, the VMM 112, in an embodiment, requests the processor 118 to deliver the pending event to the VM. In another embodiment, the VMM 112 may deliver the event to the VM by emulating the delivery of the event in software (e.g., for interrupts, by accessing the VM's interrupt descriptor table, performing access and fault checks, etc.) and performing a VM entry to the guest such that execution begins with the first instruction of the VM's event handler. In yet another embodiment, the VMM 112 makes a scheduling decision based on the indication of an open the event window, as discussed above.

[0036] **Figure 2** is a flow diagram of one embodiment of a process 200 for facilitating recognition of an open event window during operation of guest software. The process may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, microcode, etc.), software (such as that run on a general purpose computer system or a dedicated machine), or a combination of both. In one embodiment, process 200 is performed by event window monitoring logic 122 of **Figure 1**.

[0037] Referring to **Figure 2**, process 200 begins with processing logic receiving a VM entry request from a VMM (processing block 202). In one

embodiment, the VM entry request is received via a VM entry instruction executed by the VMM.

[0038] Next, processing logic transitions control to the VM (processing block 204) and determines whether the VM entry request is associated with a VMM request to be informed of an open event window (decision box 206). In one embodiment, processing logic determines whether the VM entry request is associated with a VMM request to be informed of an open event window by determining whether a pending event indicator is set to a delivery value.

[0039] If the VM entry request is not associated with a VMM request to be informed of an open event window, processing logic allows the VM to execute normally (i.e., to execute until a privileged event causes a VM exit, not shown in **Figure 2**) (processing block 208).

[0040] If the VM entry request is associated with a VMM request to be informed of an open event window, processing logic performs an event window check to determine whether the VM's event window is open (e.g., for a pending interrupt, whether its interrupt window is open) (decision box 210). In one embodiment, processing logic performs the event window check before the VM executes any instructions. Alternatively, processing logic performs the event window check after the VM executes its first instruction.

[0041] If the event window check indicates that the VM's relevant event window is open, processing logic generates a VM exit (processing block 212) and, in one embodiment, informs the VMM (e.g., by providing a relevant reason code) that the VM exit is caused by an open event window.

[0042] If the event window check indicates that the VM's relevant event window is closed, processing logic allows the VM to execute an instruction (processing block 214) and returns to decision box 210 to perform the event window check again. This loop is repeated until processing logic determines that the VM's event window is open, or until another VM exit occurs (e.g., due to the VM's access of privileged state).

[0043] In some embodiments, the execution of a guest instruction may be interrupted by a fault (e.g., a page fault, a general protection fault, etc.). Some faults may cause a VM exit. For example, if the VMM requires VM exists on page faults to protect and virtualize the physical memory, then a page fault occurring during the execution of an instruction in the VM will result in a VM exit. Hence, a VM exit caused by a fault may occur prior to performing the event window check. In addition, certain conditions (e.g., INIT, SMI, etc.) may be evaluated after the execution of the VM instruction but before performing the event window check if these conditions have higher priority than the event window check. Some of these conditions may also cause a VM exit that will occur before the event window check is performed. When the VMM receives control due to a VM exit caused by an event other than the opening of the event window, the VMM may either check the state of the VM's event window itself or request a VM entry and wait for the processor's indication that the event window is opened, as will be discussed in more detail below.

[0044] In one embodiment, pending event functionality has been implemented for interrupts by augmenting logic of a hardware reorder buffer (ROB). **Figure 3** illustrates simplified operation of ROB logic, according to one embodiment of the present invention.

[0045] Referring to **Figure 3**, ROB logic 300 operates by receiving an interrupt window state signal 302, a VMM/VM interrupt control signal 304 and an interrupt signal 306. The interrupt window state signal 302 indicates whether the interrupt window of a currently operating VM is open. The interrupt signal 306 signals that a hardware interrupt is pending. For example, this signal may be tied directly or indirectly to the processor's hardware interrupt request (INTR) pin. In one embodiment, the interrupt signal 306 is provided by the interrupt controller coupled to the processor. The VMM/VM interrupt control signal 304 indicates whether interrupts are controlled by the VMM or the VM. In one embodiment, the VMM/VM interrupt control signal 304 is maintained by the VMM and stored in the VMCS.

[0046] If the VMM/VM control signal 304 indicates that the current interrupt is controlled by the VMM and the interrupt signal 306 signals an interrupt, the ROB logic 300 asserts a signal 310 indicating that a VM exit should occur due to the interrupt. This signal may trigger microcode, dedicated hardware or software to facilitate such a VM exit.

[0047] If the VMM/VM control signal 304 indicates that the current interrupt is controlled by the VM, the interrupt signal 306 signals an interrupt,

and the interrupt window state signal 302 indicates that the VM's interrupt window is open, then the ROB logic 300 asserts a signal 312 indicating that an interrupt should be delivered to the VM. This signal may trigger microcode, dedicated hardware or software to facilitate the delivery of the interrupt to the VM.

[0048] In addition, the augmented ROB logic 300 receives as input a pending interrupt signal 308 that indicates whether there is a pending interrupt that should be delivered to a VM by the VMM. In one embodiment, the pending interrupt signal 308 is maintained by the VMM and stored in the VMCS. If the pending interrupt signal 308 indicates the existence of a pending interrupt and the interrupt window state signal 302 indicates that the VM's window is open, the ROB logic 300 asserts a signal 314 indicating that a VM exit should be generated due to an open interrupt window. This signal may trigger microcode, dedicated hardware or software to facilitate such a VM exit.

[0049] Table 1 provided below summarizes the inputs and outputs of ROB logic 300. In the embodiment shown, VM exits due to HW interrupts have priority over VM exits due to an open interrupt window. Both VM exit sources have priority over the delivery of interrupts to the VM. Other embodiments may have a different prioritization of these events. In the table, an "X" indicates a don't-care value (it may be 1 or 0); a 1 in the table indicates that the signal is asserted.

TABLE 1

ROB logic (300) Inputs				ROB logic (300) Outputs		
IntCont (signal 304) 1=VMM 0=VM	HW Int (signal 306) 1=Int Asserted	Pend. Int (signal 308)	Win. Open (signal 302)	VM-exit-int (signal 310)	VM-exit-OIW (signal 314)	Deliver int to VM (signal 312)
X	0	0	X	0	0	0
0	0	1	0			
0	1	1	0			
0	1	0	0			
1	0	1	0			
1	1	X	X	1	0	0
0	0	1	1	0	1	0
0	1	1	1			
1	0	1	1			
0	1	0	1	0	0	1

[0050] **Figure 4** is a flow diagram of one embodiment of a process 400 for determining the status of an event window of a VM. The process may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, microcode, etc.), software (such as that run on a general purpose computer system or a dedicated machine), or a combination of both. In one embodiment, process 400 is performed by a pending event module 130 of **Figure 1**.

[0051] Referring to **Figure 4**, process 400 begins with processing logic identifying a need to be informed of an opening of a VM event window (processing block 402). As discussed above, such information may be needed

to determine when to deliver a pending event to a VM, to make a scheduling decision, or for any other reason.

[0052] At processing block 404, processing logic sets a pending event indicator to a delivery value to indicate an intent to receive notification of an opening of a VM event window. In one embodiment, the pending event indicator is contained in the VMCS.

[0053] At processing block 406, processing logic requests the processor to transition control to the VM. In an embodiment, the request to transition control is sent to the processor by execution of a VM entry instruction.

[0054] Subsequently, processing logic receives control at a VM exit from the VM (processing block 408) and determines (e.g., based on a reason code associated with the VM exit) whether the VM exit was due to the opening of the VM's event window (decision box 410). If so, processing logic resets the pending event indicator to a non-delivery value (processing block 412) and performs an operation requiring an open event window (e.g., delivery of a pending event to the VM) (processing block 414).

[0055] If the VM exit was not caused by the opening of the event window, processing logic handles the VM exit according to its cause (processing block 416), and returns to processing block 408 to request a VM entry to the VM.

[0056] **Figure 5** is a flow diagram of one embodiment of a process 500 for delivering a pending event to a VM. The process may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic,

programmable logic, microcode, etc.), software (such as that run on a general

purpose computer system or a dedicated machine), or a combination of both.

In one embodiment, process 400 is performed by a pending event module 130

of **Figure 1**.

[0057] Process 500 begins with processing logic identifying an event

that should be delivered to a VM (processing block 502) and determining

whether the corresponding event window of this VM is open (processing

block 504). For example, if the event that should be delivered to the VM is an

interrupt, processing logic checks the state of the VM's interrupt window.

[0058] If the event window is open, processing logic, in one

embodiment, requests the processor to deliver the event to the VM

(processing block 506). This request may be part of a VM entry request

initiated by execution of a VM entry instruction. In another embodiment, the

VMM may emulate the delivery of the event to the VM using software

techniques and perform a VM entry to the VM such that the first instruction to

be executed will be the first instruction of the VM's event handler.

[0059] If the event window is not open, processing logic sets a pending

event indicator to a delivery value for the pending event (processing block

508). For example, if there is a single bit in the VMCS for the pending event

indicator, processing logic may set it to an asserted value (e.g., set to 1). Next,

processing logic sends a VM entry request to the processor (processing block

510). In an embodiment, the pending event indicator is a bit in the VMCS. In

an embodiment, the VM entry request is sent to the processor by execution of a VM entry instruction.

[0060] Subsequently, processing logic receives control at a VM exit from the VM (processing block 512) and determines (e.g., based on a reason code associated with the VM exit) whether the VM exit was due to the opening of the VM's event window (decision box 514). If so, processing logic resets the pending event indicator to a non-delivery value (processing block 518) and requests the processor to deliver the pending event to the VM (processing block 516). This request may be part of a VM entry request initiated by execution of a VM entry instruction. In another embodiment, the VMM may emulate the delivery of the pending event to the VM using software techniques. If the VM exit was not caused by the opening of the event window, processing logic handles the VM exit according to its cause (processing block 520), and, in one embodiment, returns to processing block 510 to request a VM entry to the VM. In response to this request, the processor transitions control to the VM and checks whether the VM's event window is open before the first VM instruction executes.

[0061] In another embodiment, processing logic determines whether the VM's event window is open after handling the VM exit at processing block 520, and returns to processing block 510 upon determining that the VM's event window is closed. Processing logic may make this determination by examining relevant state of the VM. For example, for a pending interrupt, processing logic may examine the IF bit in the EFLAGS register, as well as

other state indicating the state of the interrupt window in the VM. In this embodiment, the processor responds to the VM entry request by transitioning control to the VM, allowing the VM to execute its first instruction and then checking whether the VM's event window is open.

[0062] In one embodiment, processing logic performs an additional check to determine whether the VM's event window is open only if the preceding VM exit was caused by an event having a higher priority than the opening of the event window. In other words, processing logic performs an additional event window check only if it is possible that the opening of the event window was unnoticed by the processor due to a higher priority event that caused a VM exit.

[0063] As discussed above, in an alternative embodiment, a VMM may utilize the pending event indicator to facilitate functionality other than delivery of pending events which require an open event window. For example, such a VMM may utilize information regarding the status of a VM's event window to make VM scheduling decisions (e.g., the VMM may choose not to switch to another VM at a scheduling point if the current VM has a closed interrupt window). Such a VMM may utilize the pending event indicator to generate a VM entry to the VM that will execute until the VM has an open event window, at which time a VM exit will be generated and control will transition to the VMM. Following this VM exit, the VMM may choose to schedule another VM to execute.

[0064] Thus, a method and apparatus for facilitating recognition of an open event window during operation of guest software in a virtual machine environment have been described. It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reading and understanding the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.